Internet Week 2013

クラウド基盤における
開発・構築・運用の自動化手法
～ Infrastructure as Code ～

株式会社インターネットイニシアティブ
齊藤 秀喜

はじめに ～本セッションの概要～

近年、新たなビジネスニーズをシステムに対して迅速に反映する「継続的デリバリー」という考え方が注目を集めています。
継続的デリバリーを実現するためには、これまで我々が試みてきた、

- 物理機器の構築/仮想マシンの作成
- OS・ミドルウェアの設定
- 構築したインフラに対して実施するテスト

といった一連の構築プロセスを自動化して統合することによるスピーディーで効率的なインフラの構成管理手法が求められています。本セッションでは、具体的なツールを取り上げながら、Infrastructure Codeへの取り組み方の一例を紹介します

本セッションの目次

- 自己紹介
- Infrastructure as Code
  - Step 0 手順書
  - Step 1 スクリプト
  - Step 2 プロビジョニングツール/CloudOS
  - Step 3 協調動作
  - Step 4 未来に向けて
- 本セッション全体のまとめ

自己紹介

- 名前
  - さいとう ひでき
  - TwitterID: @saito_hideki
  - Blog: http://d.hatena.ne.jp/pyde/
- 所属
  - 株式会社インターネットイニシアティブ
  - 日本OpenStackユーザ会 ボードメンバー
- 趣味
  - OpenStack/Ansible

自己紹介: 日常の仕事

- クラウドっぽいアーキテクチャの設計
- サーバの構築・設定
- ネットワーク機器の設定
- ストレージ機器の設定
- 実証実験設備を静かに運用
- 必要に応じて小道具や大道具の開発
- 本業の合間に研究開発

自己紹介: 研究開発用コンテナDCの管理人

- コンテナデータセンタに関するさまざまな実証実験を行う基盤を管理しています

- ITサーバ: 約200台
- ネットワークスイッチ多数
- ストレージ数台
- コントローラブルなUPS数台
- 計測用センサー多数

Infrastructure as Code

Infrastructure as Codeとは?

- What?
  - 継続的デリバリを考えるうえで重要になるインフラの構成管理手法
  - 構築・設定・確認作業をプログラム(コード)によって自動化する
  - 自動化した作業を統合してシステムの構成管理を行う
  - DevOpsを進める上で必須となるエンジニアのためのLifehack

- Who? / When?
  - 出典: Web Operations 第5章
  - 著者: アダム・ジェイコブ氏
  - 出版: 2010.6
  - 日本語版:ISBN978-4-87311-493-4

Infrastructure as Codeとは?

- Where?

  継続的デリバリやDevOpsの定義の中に出てくる「自動化」に関する部分を実装したものが「Infrastructure as Code」であると考えています。

The adoption of DevOps is being driven by factors such as:

- Use of agile and other development processes and methodologies
- Demand for an increased rate of production releases from application and business unit stakeholders
- Wide availability of virtualized and cloud infrastructure from internal and external providers
- Increased usage of data center automation and configuration management tools

ここ

～ wikipedia http://en.wikipedia.org/wiki/DevOps より～

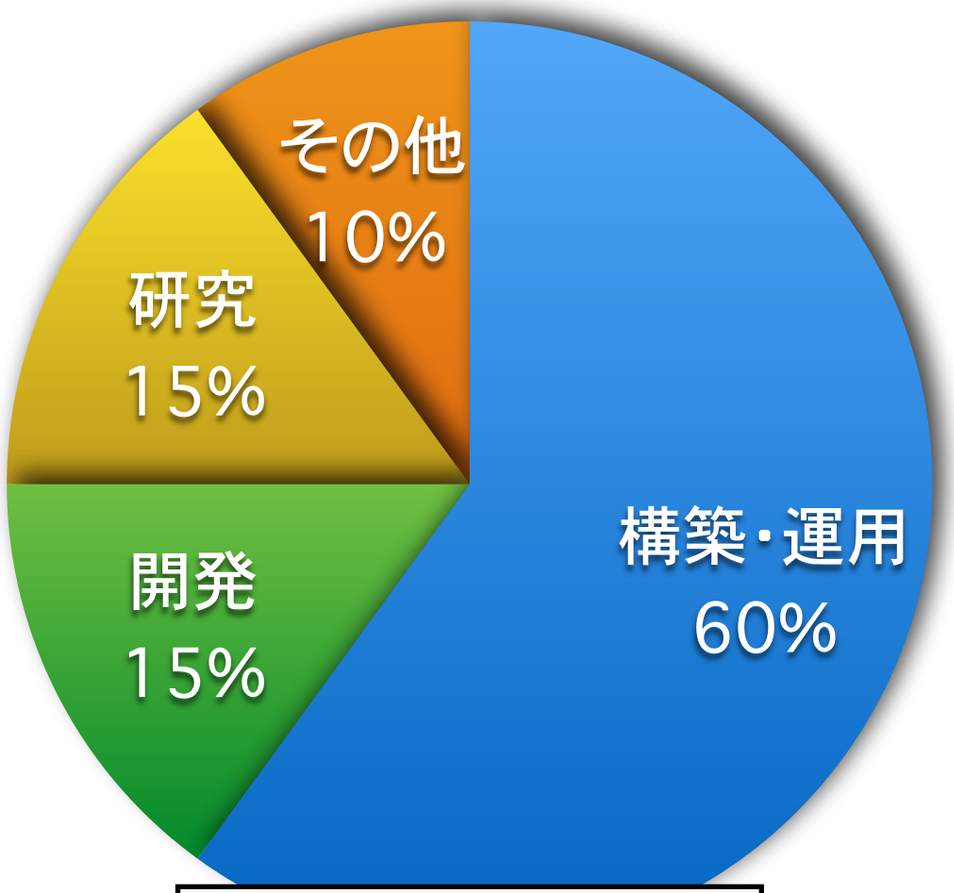
Infrastructure as Codeとは?

- Why?(Lifehack)

  構築などの単純作業は台数が増えると削られる時間が指数関数的に増大する。これを減らして、もう少しリサーチや研究・開発のための時間を増やしたい

  - お決まりの構築・運用作業に当てている時間を減らして研究と開発にあてる!
  - 作業を誰かに任せる?

How?
(どんな方法で?)



Infrastructure as Code

Step 0

手順書を書いて引き継ぐ

- 構築・確認手順書を作って仕事を誰かに引き継ぐ
- 2000年あたりから現在まで脈々と続く伝統芸能

作業者: ぼく
確認者: (失敗する度に人が追加になる)
承認者: きみ

1) Switchにsshログイン
   - UserId: foo
   - Password: hogehoge
2) VLAN100が存在していないことを確認
   > show vlans
   [確認]VLAN100が表示されない
3) VLAN100を作成する
   > set vlans VLAN100 vlan-id 100
4) VLAN100が存在していることを確認
   […]

作業手順書サンプル

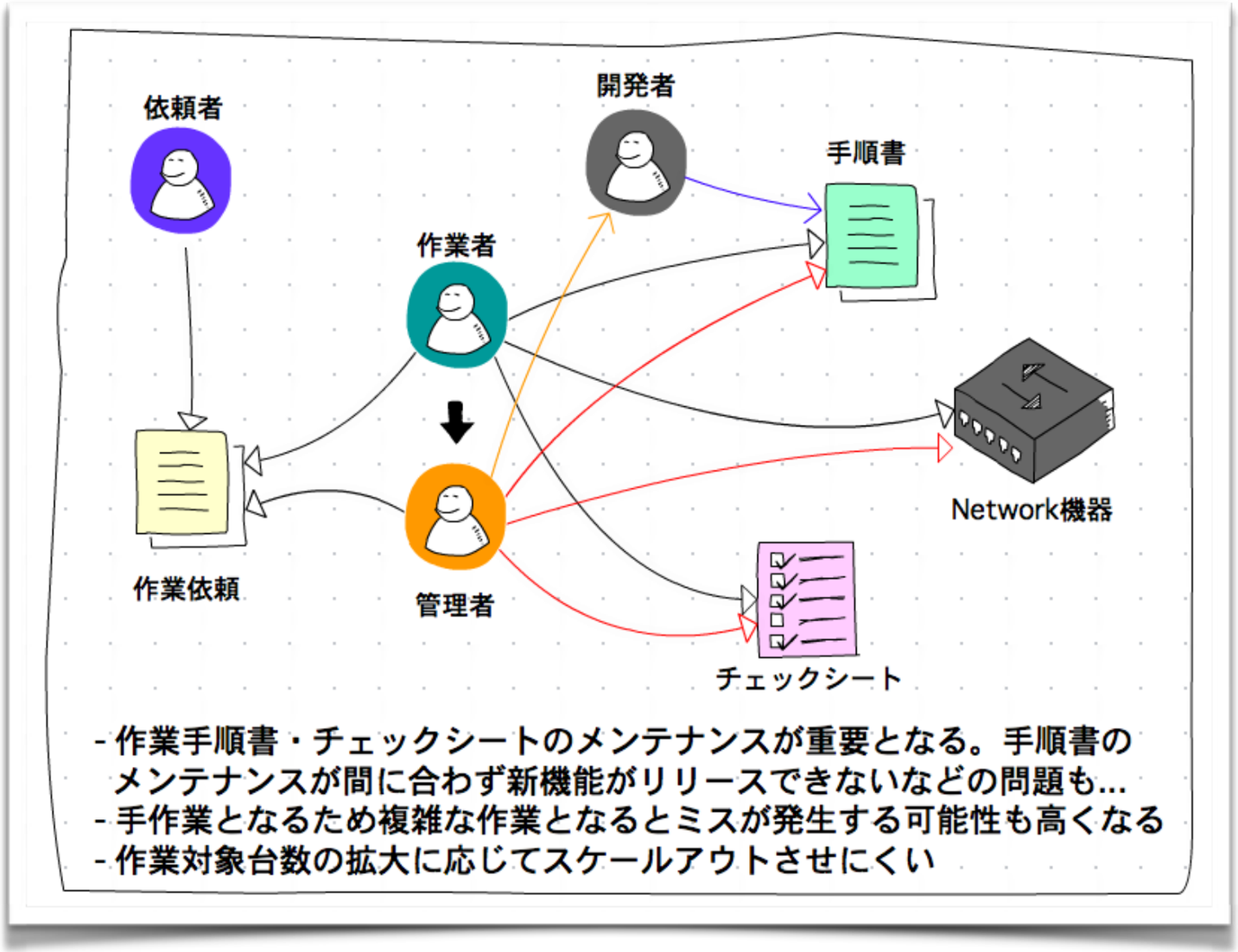
作業者: ぼく
確認者: (確認漏れがある度に人が追加になる)
承認者: きみ

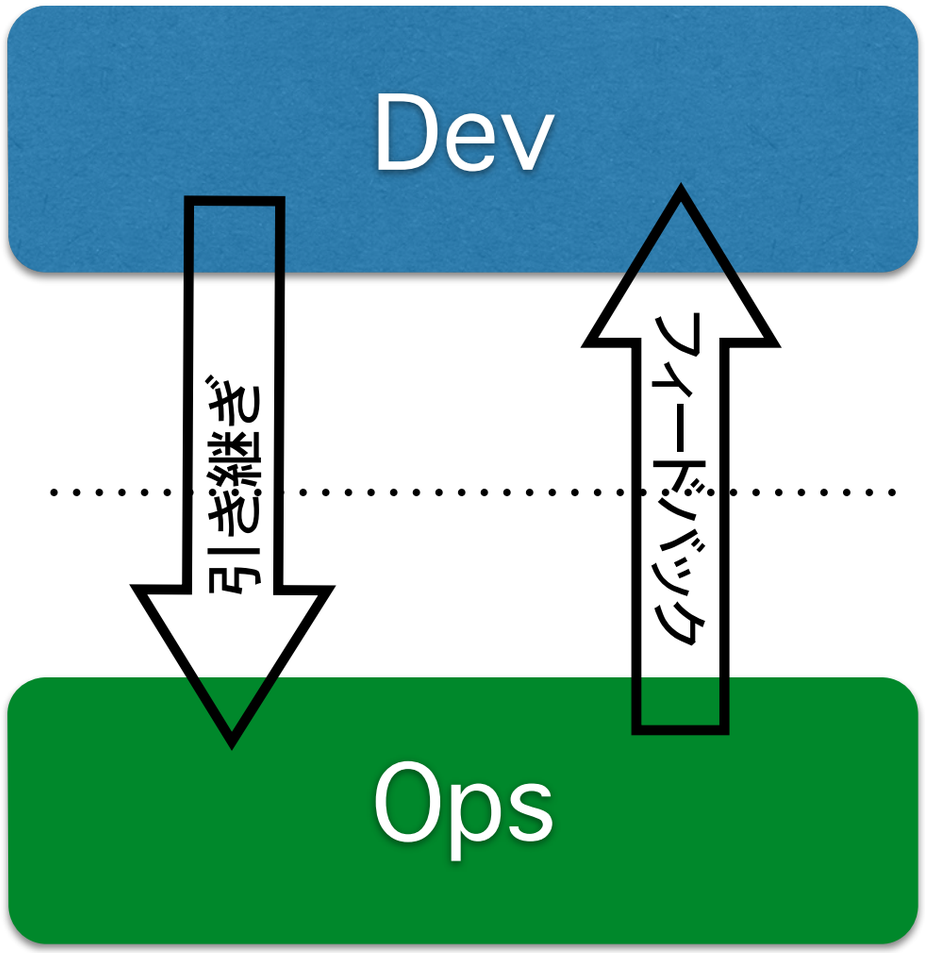
□ AがTrueであること
 - Trueでなかった場合は手順書Xを実施
□ BがFalseであること
 - Falseでなかった場合は担当者に電話
 […]

チェックシートサンプル

ワークフロー

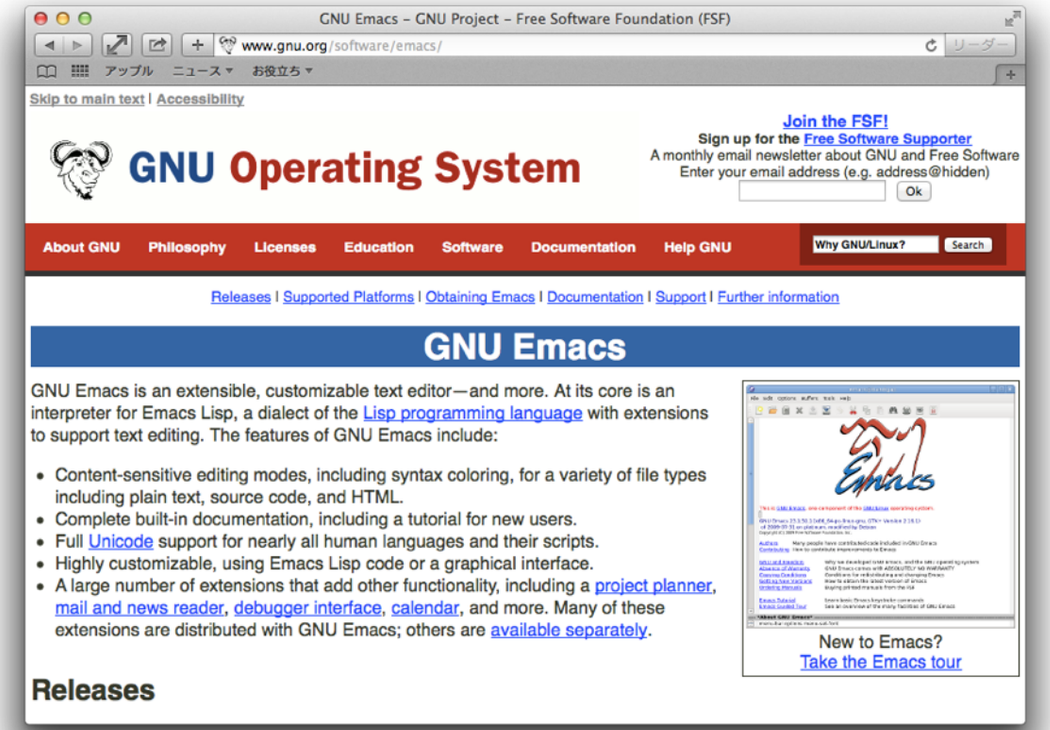
- 手順化されたことによりワークフローができる
- 手順書による開発側からの引き継ぎと運用側からのフィードバックという一連のプロセスを、いかに円滑に回すかがシステム運用のカギとなる





DevとOpsという役割ができた

ツール類

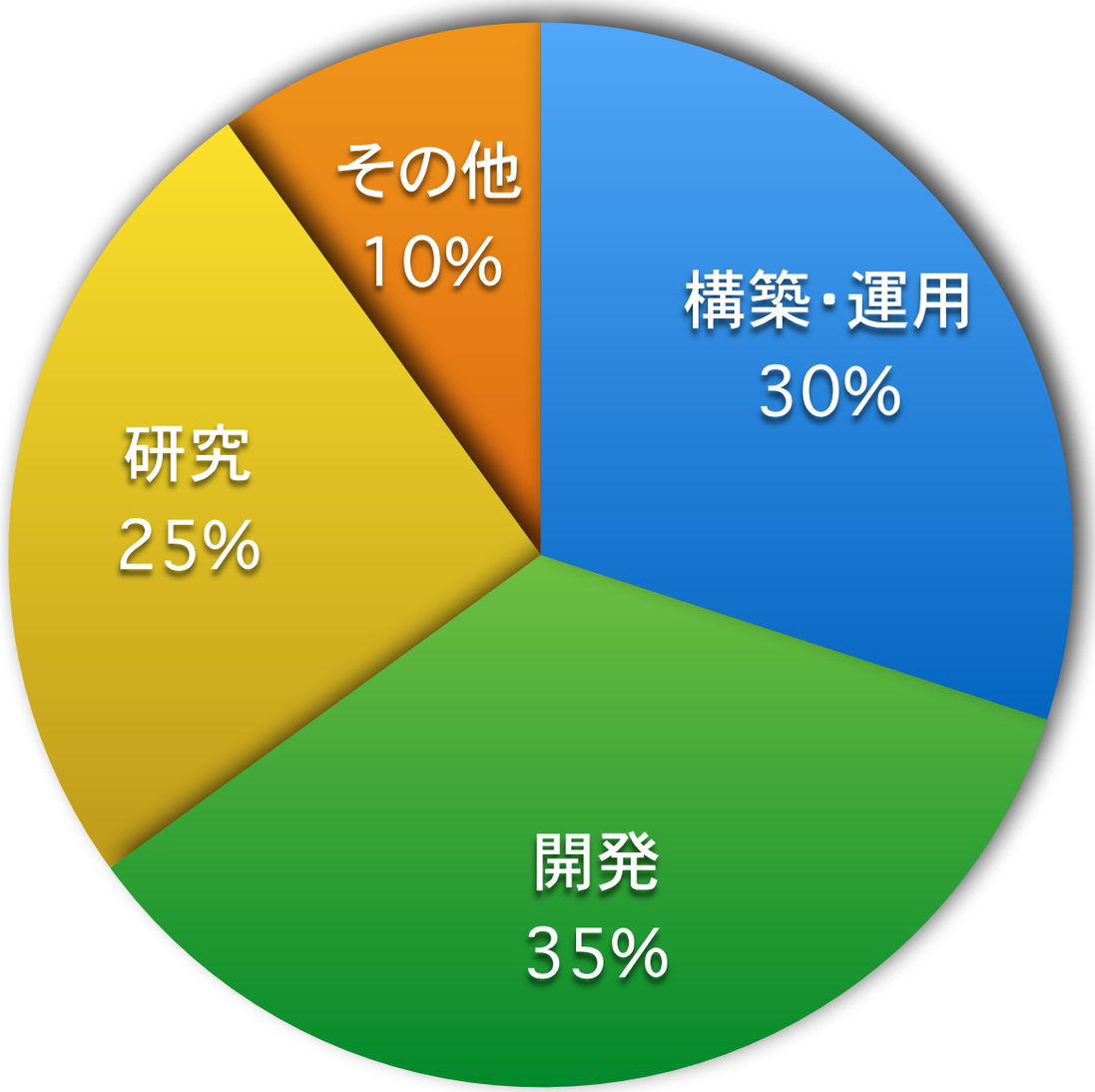


手順書方式では
Wiki化がゴールだった

Step-0 まとめ(Lifehackは成功したのか?)

手順書の作成・更新や、どうしても引き継げない作業は残るが、
それでも単純作業からは解放される。
研究開発の時間も今までより遥かに確保できた!
すばらしい! Lifehack完成!

ちょっと待て!



Step-0 まとめ(手順書方式が落とす影)

- DevとOps間のライフサイクルが滞る

  コマンドラインオプションの名称修正などのようなシステム的には小変更でも、手順書の修正箇所が多数になる場合がある。その結果修正を見送るような本末転倒な状況が発生しやすい

- より良い状態に進化させにくい

  Dev(出す)とOps(受ける)双方の負担を考えて小変更程度なら適用を見送る…というように、システムを良い状態にするための改修が行いにくい状態が発生しやすい状況となっている

- スケールしにくい

  同時に大量の設定を行うような場合は、手順書方式では人海戦術で対応するしかないため、作業日時、規模などを常に意識しなければならない状態が常時発生する

Infrastructure as Code

Step 1

作業を自動化する

- コード化

  手順書の中で、まずはプリミティブな操作部分を人ではなく機械にやらせる。
  手順書の代わりにコードを書くことによって作業を自動化してエンジニアの時間を、もっと創造的な仕事に使えるようにする。

  1) ネットワークスイッチにVLANを作成する
  2) VLANが正しく作成されていることを確認する
  3) 接続されている機器間の疎通を確認する

  というような単純作業をするコードが取り組みの第一歩。

  細々と個々人で作っていたけれど、本当に真面目に考え始めたのは、だいたい2003年頃でしょうか

Step-1では作業のスクリプト化などにより自動化に取り組む

作業の種類

我々が行う作業には大きく2種類あります。

1. 既存の基盤の設定を変更する(設定変更)

   定められたお作法にしたがって人間が行っているオペレーションをコード化する　Infrastructure as Code: Step-1

2. 新たに基盤を構築する(新規構築)

   定められたお作法の中にコード化しにくいものがある

   例えば、

   - Power Off/On
   - OSインストールDVDの交換やEject
   - OSインストール作業中のコンソールオペレーション
   - 仮想化基盤の構築や仮想マシン・ネットワーク・ストレージの管理

   などなど。　Infrastructure as Code: Step-2

設定変更作業手順書のコード化

既存の基盤を設定変更する手順書をコード化してみる。

作業者: ぼく
確認者: (失敗する度に人が追加になる)
承認者: きみ

1) Switchにsshログイン
   - UserId: foo
   - Password: hogehoge
2) VLAN100が存在していないことを確認
   > show vlans
   [確認]VLAN100が表示されない
3) VLAN100を作成する
   > set vlans VLAN100 vlan-id 100
4) VLAN100が存在していることを確認
   […]

作業手順書サンプル

どのような手段があるだろうか?

案1: CLI
- a) Tclでexpect
- b) Perlでexpect
- c) Pythonでpexpect
- d) Rubyでpty & expect

案2: SNMP
SNMPが使えるならwriteする

案3: NETCONF
SSH経由でNETCONFでwriteする

手段を決めたらコードを書く

```
class JunosVlan(object):
    def __init__(self, module):
        self.module = module
        self.vlan_name = module.params['vlan_name']
        self.vlan_id = module.params['vlan_id']
        self.vlan_desc = module.params['vlan_desc']
        self.state = module.params['state']
        with manager.connect(host=node,
                             port=port,
                             username=user,
                             password=passwd,
                             hostkey_verify=False) as m:
            self.config = m.get_config(source='candidate').data_xml

    def push(self, config):
        try:
            with manager.connect(host=node,
                                 port=port,
                                 username=user,
                                 password=passwd,
                                 unknown_host_cb=_always_unknown_true) as mgr:
                mgr.edit_config(target="candidate",
                                config=config,
                                test_option="test-then-set")
                mgr.commit()
                rc = 0
                out = 'commit succeeded'
                err = ''
        except TimeoutExpiredError as e:
            rc = 1
            out = 'commit failure'
            err = 'operation timeout'
        return (rc, out, err)

    def vlans(self):
        elm = ElementTree.fromstring(self.config)
        elm_vlans = elm.find('.//vlans')
        vlan_info = dict()
        for elm_vlan in elm_vlans.findall('.//vlan'):
            vlan_id = elm_vlan.findtext('.//vlan-id')
            name = elm_vlan.findtext('.//name')
            description = elm_vlan.findtext('.//description')
            vlan_info[name] = dict(vlan_id=vlan_id, description=description)
        return vlan_info.keys()

def _always_unknown_true(host, fingerprint):
    return True
```

junos_vlan.py

作業者: ぼく
確認者: (失敗する度に人が追加になる)
承認者: きみ

1) VLAN-IDを指定してjunos_vlan.py を実行
2) コマンド実行結果がsuccessfulであることを確認する

作業手順書サンプル

作業が大幅削減！すばらしい！
Lifehack完成！

しかし、ここまで来たら…
もうちょっとがんばれるかも！

手順書をコードにする…

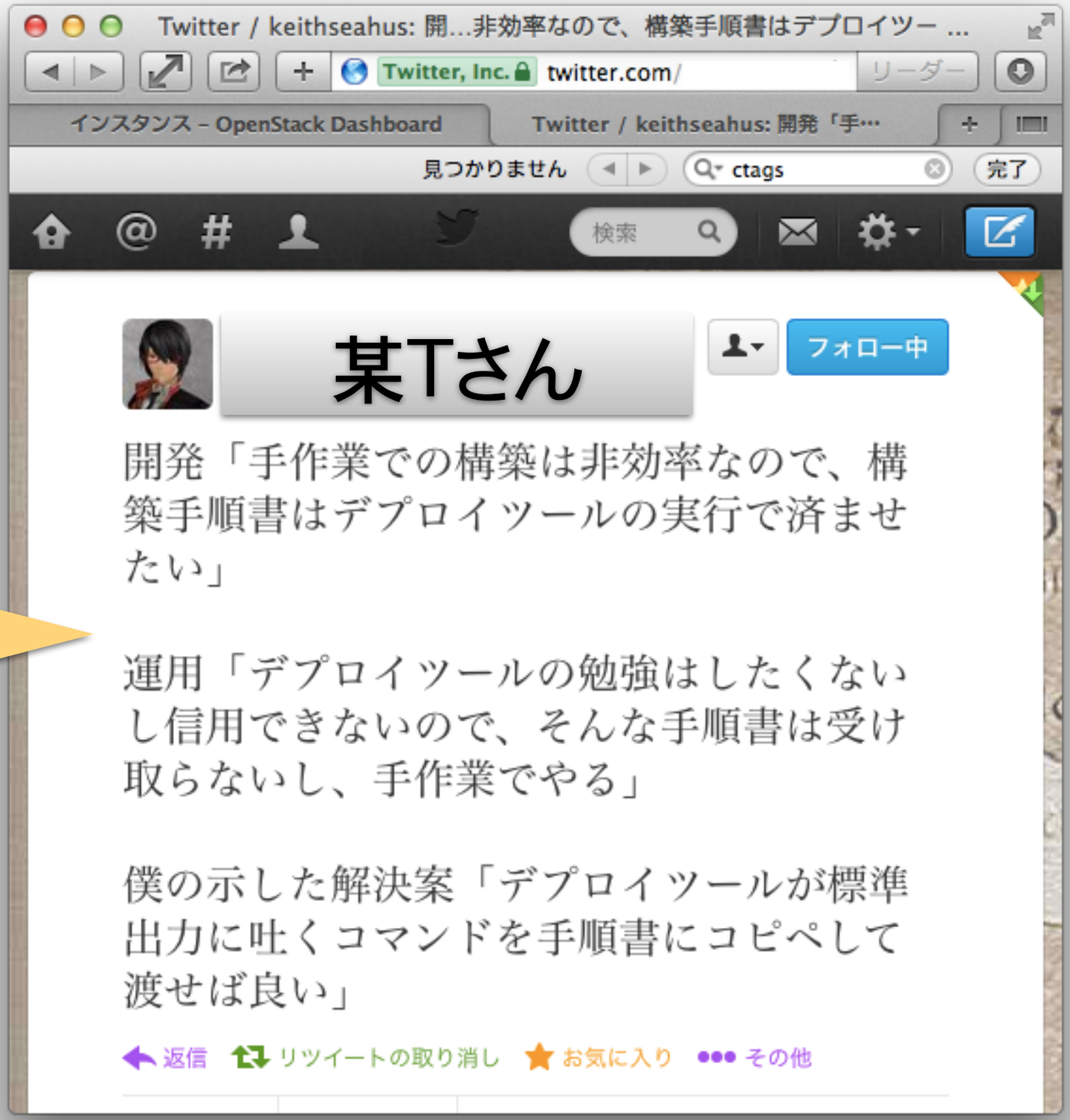
単純作業をコードにする…現在もある程度実践しているはずです。新しい言葉で定義しただけなんじゃ…ちょっと今更感が…

Step-1 まとめ(Lifehackは成功したのか?)

- 手順書に記載されている作業をコード化した
- Step-0で創出した時間を有効に使って、Opsにお願いしている作業をコード化して自動化への移行を進めることができる
- 正確にコマンドを叩き、その結果を正確に読み取り、判断して次の手順を実行する…というような人間よりもコンピュータが得意とする部分は、コード化したほうがDevもOpsも幸せになれる

Step-1 余談(現場の反応は予想の斜め上だったことも)

同じ体験をしたことがありますどこも一緒?:)



Infrastructure as Code

Step 2

Step-2: インフラ構築作業をコードにする

- 物理サーバ構築・仮想マシン構築の自動化
  1) CD/DVDを入れ替えつつ手作業で行っていたのでは、まったくスケールしない。またスピード感も世の中と乖離してしまう
  2) 手作業でコツコツと行っていたサーバインストール作業を自動化
  3) 仮想マシンへのOSインストールも1)と同じ方式で可能

  物理サーバは2002年あたり、仮想マシンは2008年あたりから取り組みをはじめた

- クラウド基盤管理の自動化

  2009年あたりから本格的に各社が対応を開始した物理サーバと仮想マシンの集合体であるクラウド基盤では、仮想マシン・ストレージ・ネットワーク・IPアドレスなどのリソース管理の自動化、それらをユーザやテナントなどの管理単位でアイソレーションできる機能の管理が求められる

  クラウド基盤を管理する仕事は多岐にわたるため、これを手作業で行うのは絶望的。何らかの仕組みを導入する必要がある

Step-2ではインフラの構築作業と管理の自動化に取り組む

物理サーバ・仮想マシン構築の自動化

～実証実験コンテナで採用したプロビジョニングシステム～

Cobblerによるベアメタルプロビジョニング(1)

- **物理サーバ構築の変遷**

  手作業による初期構築のコストが大きく台数の増加に対応できない

CD/DVDメディアから手作業でインストール&初期設定

kickstartを利用した半自動インストール&初期設定

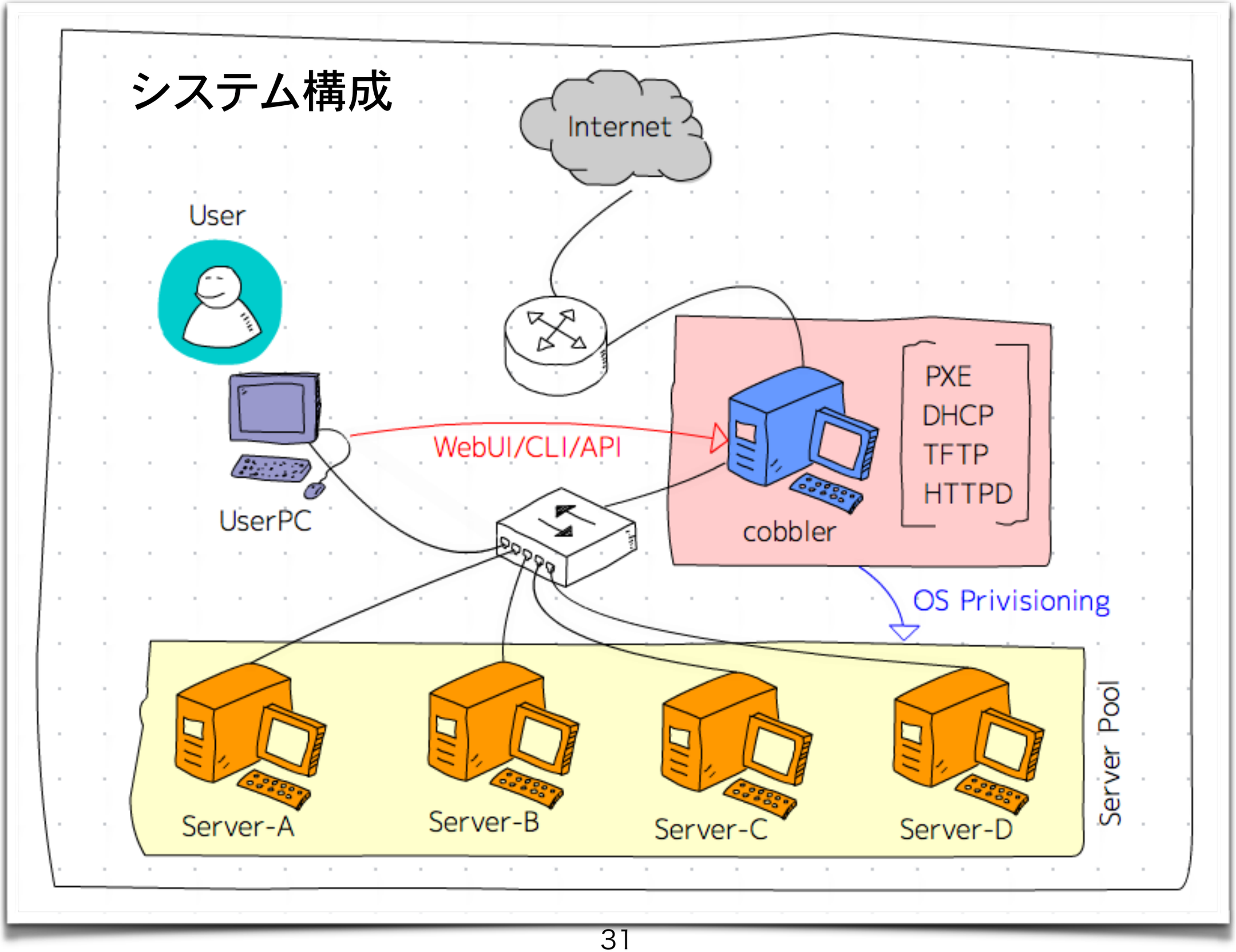
PXE/DHCP/TFTP/NFS/kickstartを利用した
自動インストール&初期設定

更に高度な自動化を目指してcobblerの利用を開始

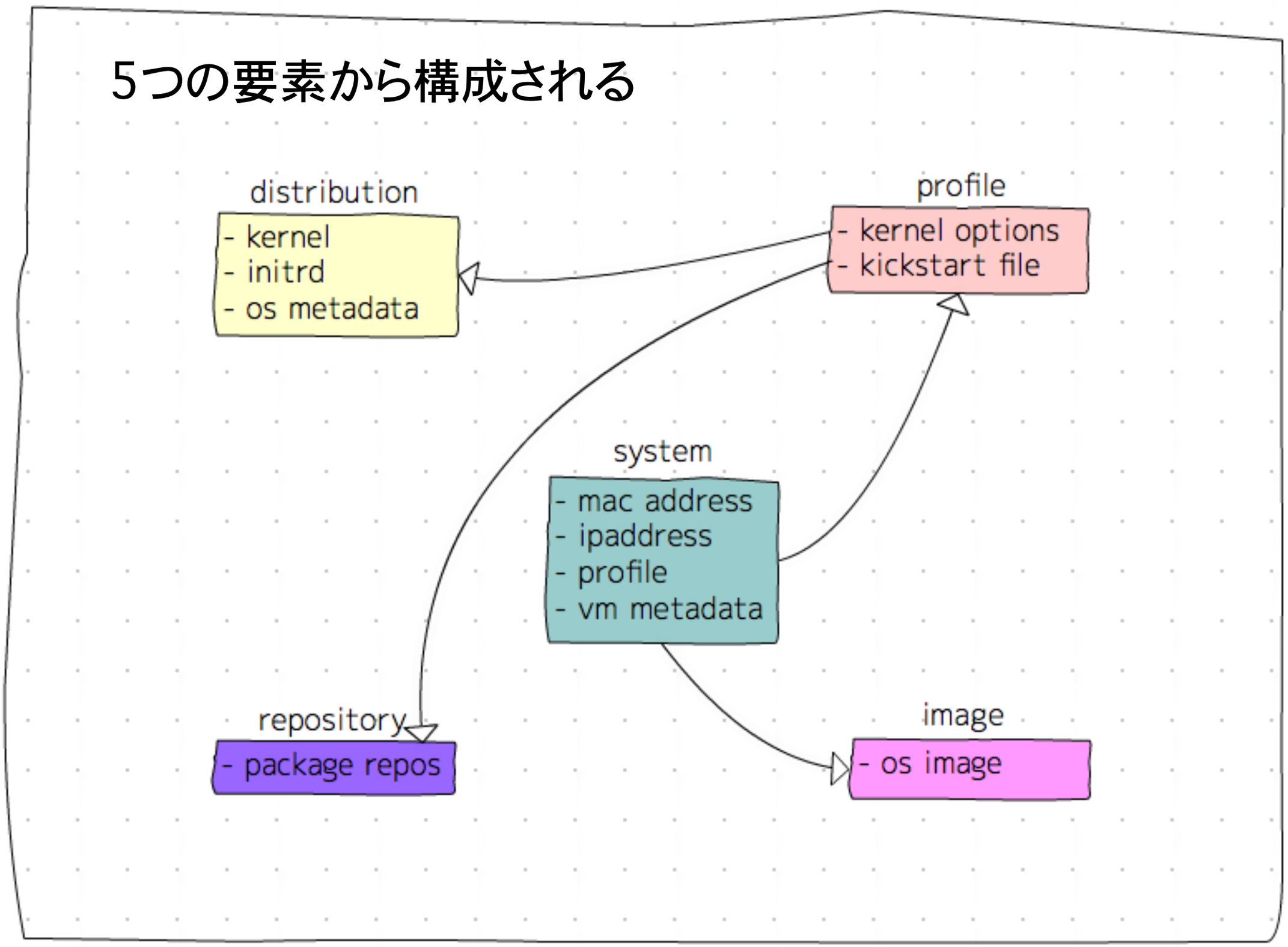
Cobblerによるベアメタルプロビジョニング(2)

- What?/Why?
  - サーバへのOSインストール作業を自動化する仕組み
  - 手軽にOSのネットワークインストール環境を構築できる
  - CLI/WebUIの他に外部から制御可能なAPIを持っている
  - PXE/DHCP/TFTPという枯れきって安定した技術を採用している
  - Red Hat系/Debian/Ubuntuなど多くのディストリビュージョンでパッケージが提供されていて導入しやすい
  - 物理サーバ・仮想マシンどちらにもOSをデプロイすることが可能
- Who?/When?/Where?
  - 公式サイト: http://www.cobblerd.org/
  - 開発を支える多数のサポート団体が存在する
    http://www.cobblerd.org/supporters.html
  - github上で確認できる最初のリリース(v0.4.4)から既に7年
  - 現在(2013/11)の最新バージョンはv2.4.0
  - エンタープライズ分野でも広く利用されている
    http://www.cobblerd.org/users.html

Cobblerのシステム構成図



Cobblerの構成要素



Cobblerの導入から稼働まで(ubuntu12.04lts編 [1])

1. インストールする(aptで楽々)

```
$ sudo apt-get install -y cobbler cobbler-web
$ sudo apt-get install -y dnsmasq
```

2. 設定する(修正ポイントはとりあえず3つのみ)

1. /etc/cobbler/modules.conf

```
[dns]
module = manage_dnsmasq
[dhcp]
module = manage_dnsmasq
```

2. /etc/cobbler/dhsmasq.template

```
dhcp-range=192.168.100.100,192.168.1.199
```

3. /etc/cobbler/settings

```
manage_dhcp: 1
```

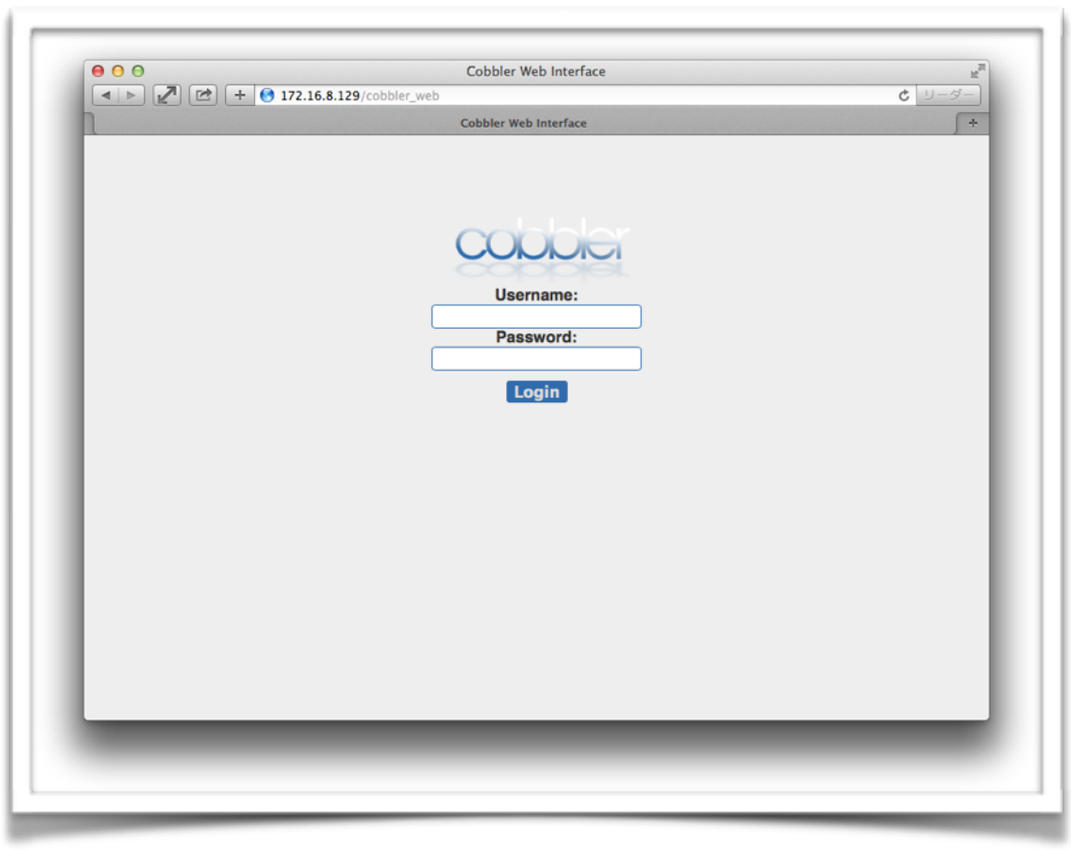
Cobblerの導入から稼働まで(ubuntu12.04lts編 [2])

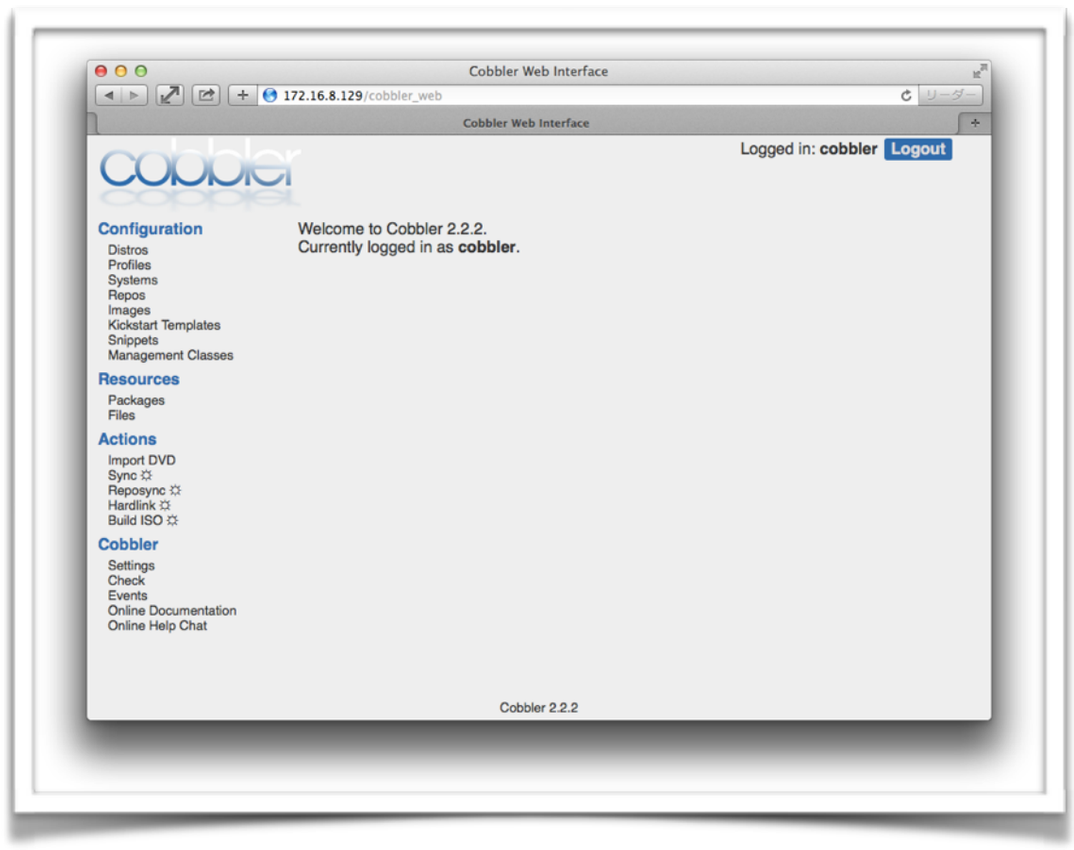
3. 起動する(cobbler syncを忘れずに!)

```
$ sudo service cobbler restart
$ sudo service apache2 restart
$ sudo cobbler sync
```

4. 接続確認する

ブラウザから http://127.0.0.1/cobbler_web にアクセス!





Cobblerの導入から稼働まで(ubuntu12.04lts編 [3])

5. 利用する(CLIから利用する[1])

1. ISOイメージのダウンロードとkickstart/seedファイル作成する
   - ISOイメージ
     - ubuntu-12.04.3-server-amd64.iso
     - CentOS-6.4-x86_64-minimal.iso
   - kickstart/seedfile
     - /var/lib/cobbler/kickstarts/ubuntu1204.seed
     - /var/lib/cobbler/kickstarts/centos64.ks
2. ISOイメージをインポートする

   インポートに成功するとdistributionとprofileが作成される

```
$ sudo mount -o loop ubuntu-12.04.3-server-amd64.iso /mnt
$ sudo cobbler import --arch=x86_64 --path=/mnt \
--name=ubuntu1204lts
$ sudo umount /mnt
$ sudo mount -o loop CentOS-6.4-x86_64-minimal.iso /mnt
$ sudo cobbler import --arch=x86_64 --path=/mnt \
--name=centos64
$ sudo umount /mnt
```

Cobblerの導入から稼働まで(ubuntu12.04lts編 [4])

3. インポート時に作成されたプロファイルを編集する
   プロファイルにはdistributionとkickstart/seedの紐付けを設定する

```
$ sudo cobbler profile edit --name=ubuntu1204lts-x86_64 \
--kickstart=/var/lib/cobbler/kickstarts/ubuntu1204.seed
$ sudo cobbler profile edit –name=centos64-x86_64 \
--kickstart=/var/lib/cobbler/kickstarts/centos64.ks
$ sudo cobbler sync
```

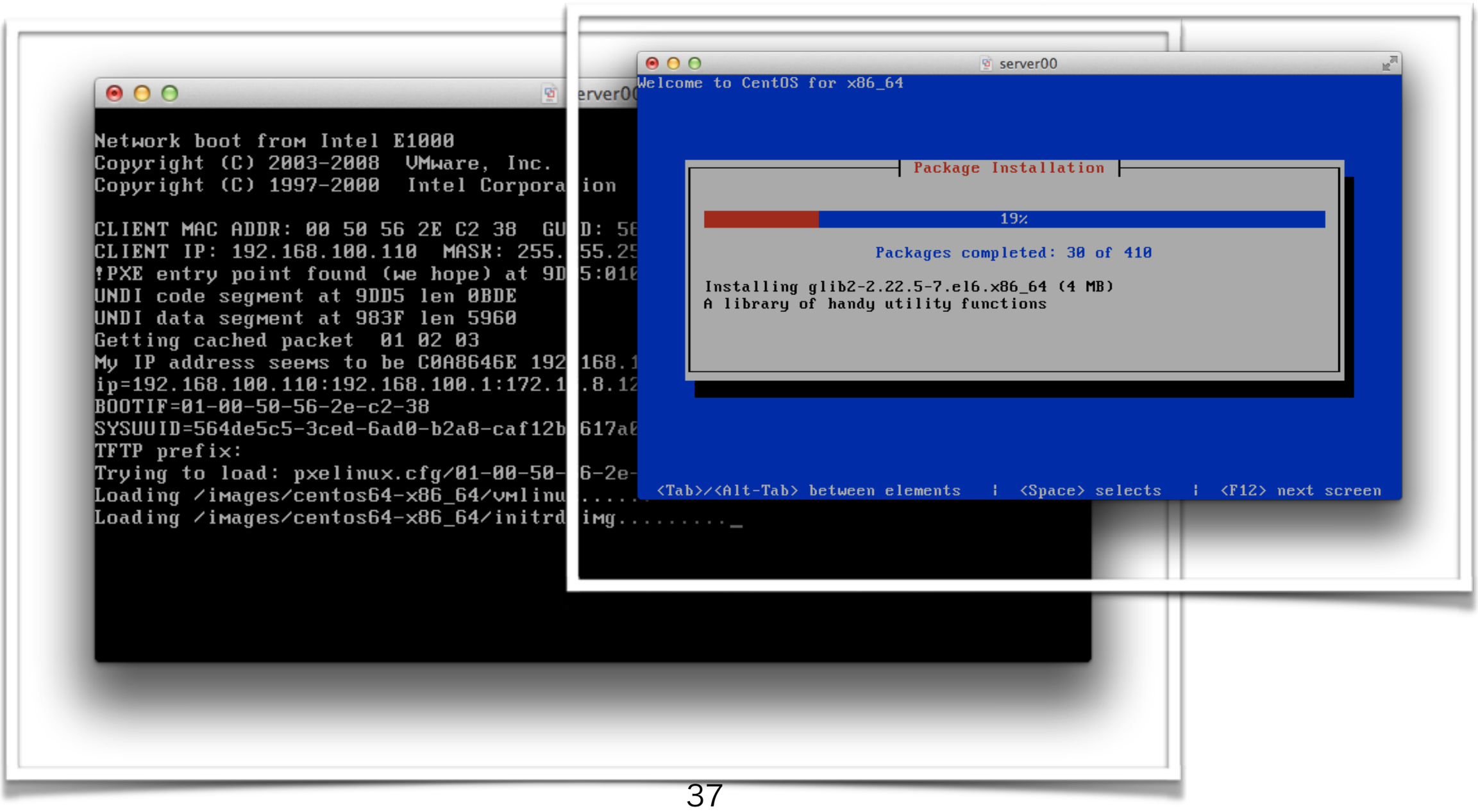
4. システム(プール内のサーバ)を登録する
   システムのMACアドレス/IPアドレスとプロファイルを紐付けを登録する

```
$ sudo cobbler system add --name=server00 \
--profile=centos64-x86_64 \
--mac=00:50:56:2E:C2:38 --ip-address=192.168.100.110
$ sudo cobbler system add --name=server01 \
--profile=ubuntu1204lts-x86_64 \
--mac=00:50:56:39:7A:F0 --ip-address=192.168.100.111
```

Cobblerの導入から稼働まで(ubuntu12.04lts編 [5])

6. プロビジョニング対象サーバを起動する

PXE/DHCP/TFTPを利用して、kickstart/seedの設定内容に従ってOSのネットワークインストールが開始される



Cobblerの導入から稼働まで(ubuntu12.04lts編 [6])

7. API経由で利用してみる

XML-RPC APIでCobblerを操作するのは意外に簡単。

例えば登録済OSイメージの情報を取得するコードを書いて実行してみる。

参考URL: https://github.com/cobbler/cobbler/wiki/XMLRPC

サンプルコード実行結果

```
- arch: x86_64
  boot_files: {}
  breed: redhat
  comment: rhel6.4
  ctime: 1384008415.302049
  depth: 0
  fetchable_files: {}
  initrd: /var/www/cobbler/ks_mirror/centos64-x86_64/images/p
  kernel: /var/www/cobbler/ks_mirror/centos64-x86_64/images/p
  kernel_options: {}
  kernel_options_post: {}
  ks_meta: {tree: 'http://@@http_server@@/cblr/links/centos64
  mgmt_classes: []
  mtime: 1384008415.342452
  name: centos64-x86_64
  os_version: rhel6
  owners: [admin]
  redhat_management_key: <<inherit>>
  redhat_management_server: <<inherit>>
  source_repos: []
  template_files: {}
  tree_build_time: 1362445555.957609
  uid: MTM4NDAwODQxNS4zMTg0ODQ0My4wMDU2OQ
```

サンプルコード

```
#! /usr/bin/env python

import xmlrpclib
import yaml

host = '172.16.8.129'
user = 'cobbler'
passwd = ''

server = xmlrpclib.Server('http://%s/cobbler_api' % host)
token=server.login(user, passwd)
distro_list = server.get_distros()

print yaml.safe_dump(distro_list)

##
## [EOF]
##
```

クラウド基盤管理の自動化

～CloudOS～

OpenStackによるクラウド基盤の管理

- ## 仮想マシンの構築・管理の変遷

  仮想マシンは物理マシンと比較すれば構築にかかる手間は最初から低かったが増加の一途をたどる管理対象ノード数が問題となる。

SolarisコンテナやOpenVZなどから手動で提供

↓

Xen/kvmのHyperVisorに移行するも依然として手動で構築

↓

管理対象VMの飛躍的な増加

↓

VM数の増加とともに管理の自動化を目指してCloudOSを導入

OpenStackによるクラウド基盤の管理

- What?/Why?
  - Amazon Web Serviceライクなクラウド環境を構築できるCloudOS
  - 仮想マシン・ストレージ・ネットワーク構成の管理を行う
  - 機能ごとに高度にコンポーネント化されており単機能だけでも利用可能
  - クラウド基盤を外部から制御可能なAPIを持っている
  - いつの間にか勝ち組になりつつあってちょっと寂しい
  - ライバルはCloudStackとEucalyptus
- Who?/When?/Where?
  - 公式サイト: http://openstack.org
  - 日本OpenStackユーザ会: http://openstack.jp/
  - コミュニティで選出されたリリースマネージャを中心としてコンポーネント毎に開発が進められている
  - 日本国内でも日本OpenStackユーザ会が中心となって、利用促進のための資料公開・勉強会開催など活動が活発
  - 2010-10の最初のリリース(Austin)以降、2013-10の最新リリース(Havana)まで8つのメジャーリリースが行われている

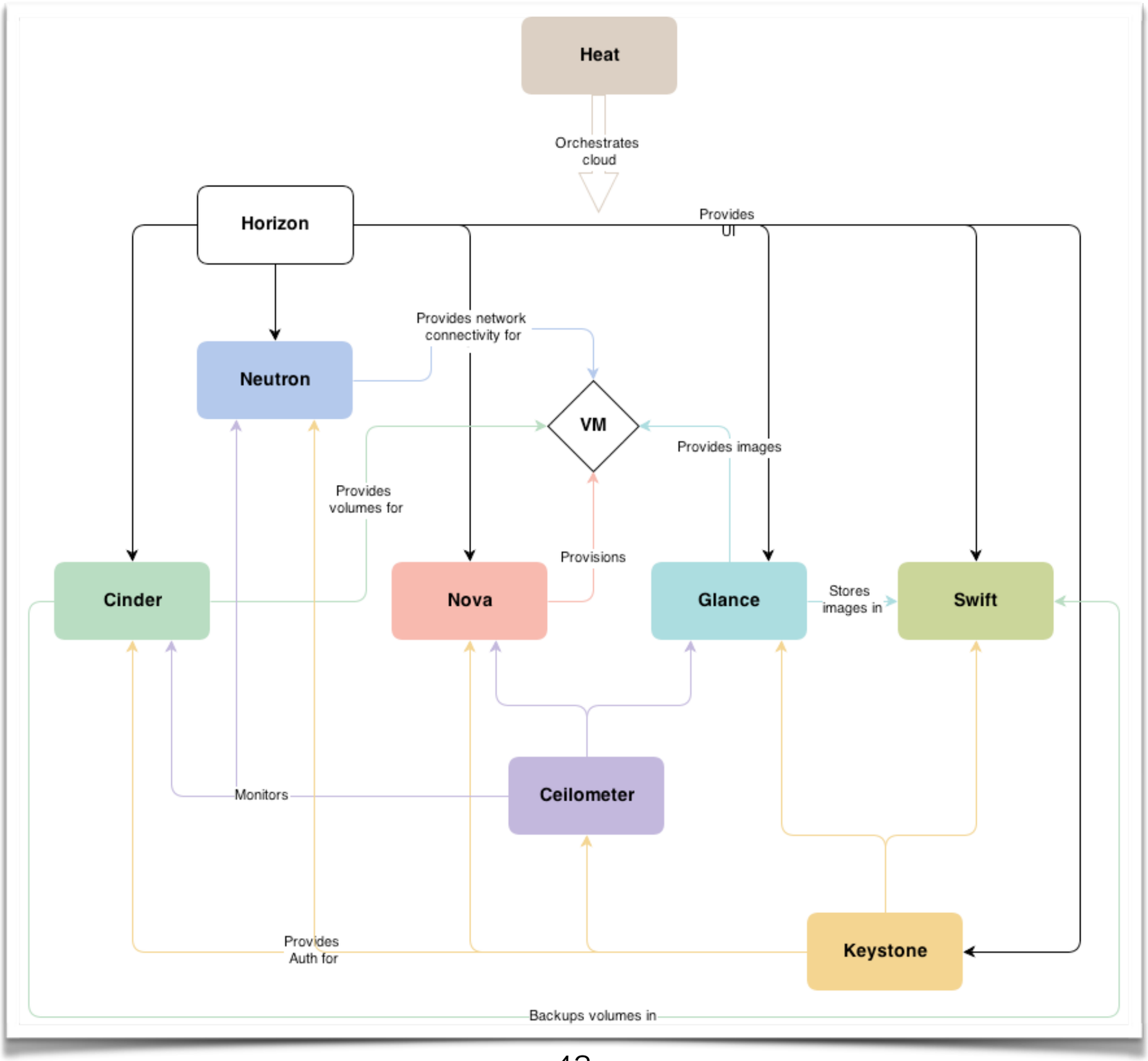
OpenStackの構成

OpenStackは機能毎にコンポーネント化され、コンポーネント間は外部APIで疎結合されている

コンポーネント名	機能
Nova	ハイパーバイザを制御し、仮想マシンの管理を行う
Glance	仮想マシンの起動イメージ・スナップショットの管理を行う
Neutron	様々なNW機器と連携し、仮想ネットワークを実現する
Cinder	仮想マシンに対してブロックストレージ機能を提供する
Swift	オブジェクトストレージ機能を提供する
Keystone	OpenStackコンポーネント全体に認証機構を提供する
Horizon	利用者にWebUIを提供する
Ceilometer	リソースの使用状況を計測し利用者に情報を提供する
Heat	仮想マシンのオーケストレーション機能を提供する

出典: http://openstack.jp/assets/files/20130803/OSC2013_Kyoto.pdf

OpenStackの構成要素の関係



OpenStackに対するOSの対応状況

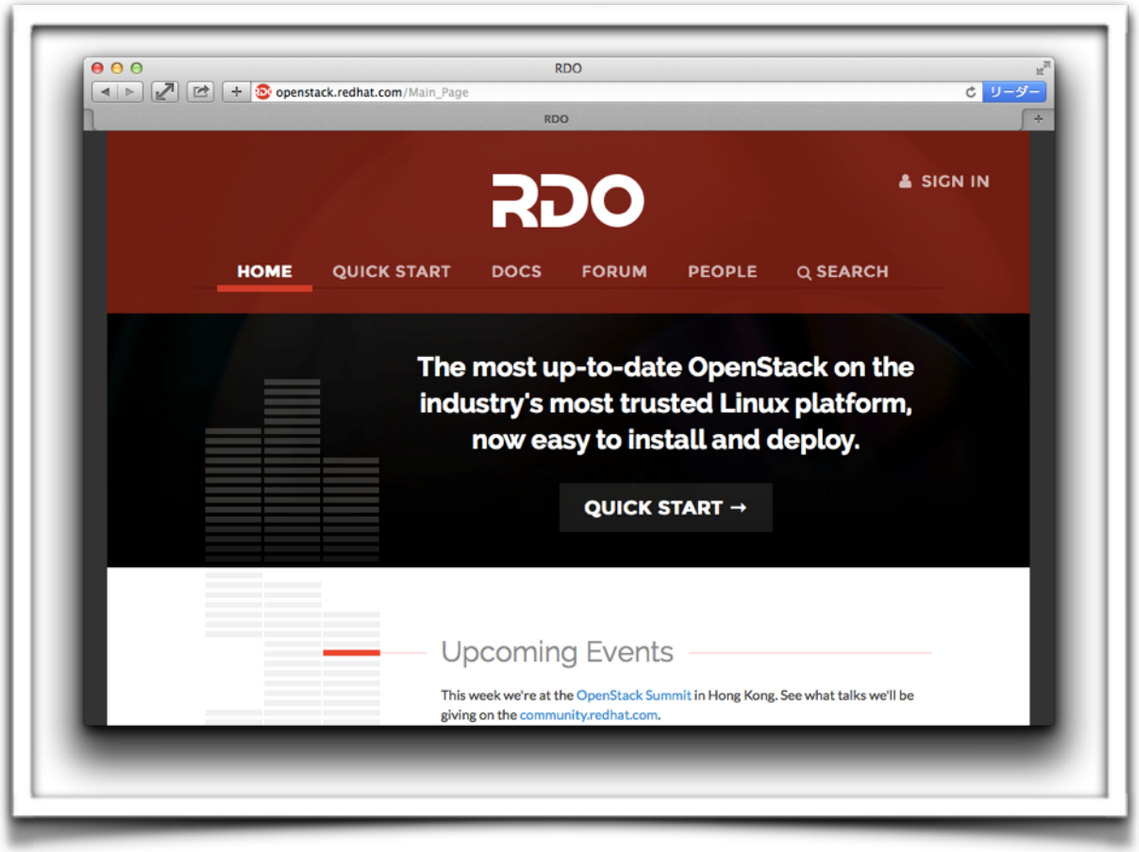
主要なディストリビューションも積極的に対応を初めている

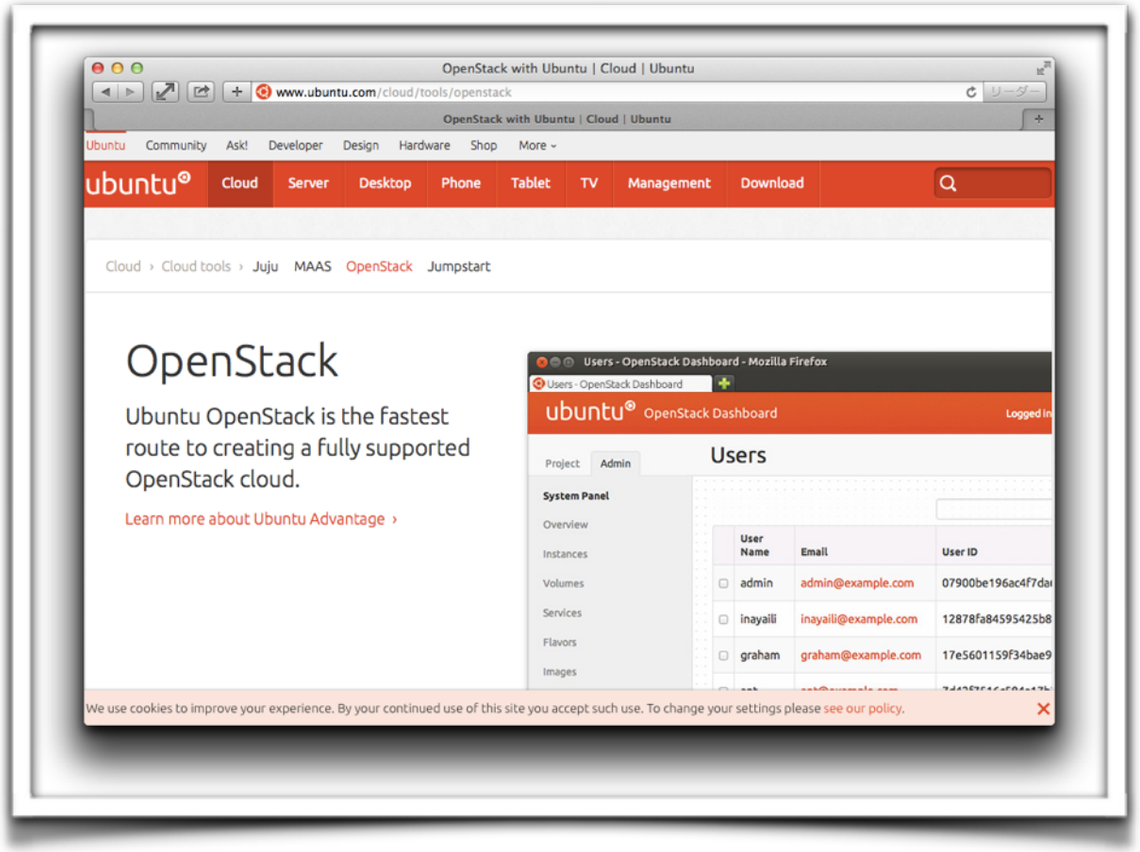
- Red Hat

  http://jp-redhat.com/openstack/rdo/

- Ubuntu

  http://www.ubuntu.com/cloud/tools/openstack





OpenStackのインストーラ

- DevStack

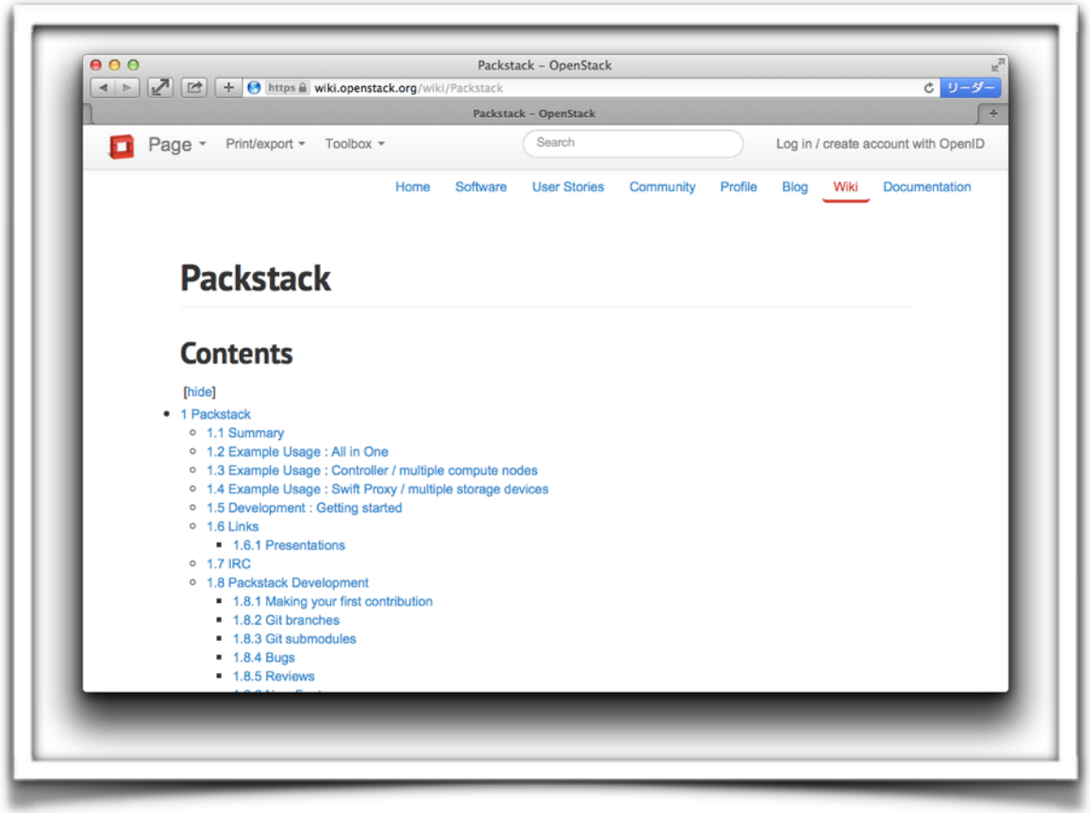
  http://devstack.org/

- Packstack

  https://wiki.openstack.org/wiki/Packstack

- openstack-ansible

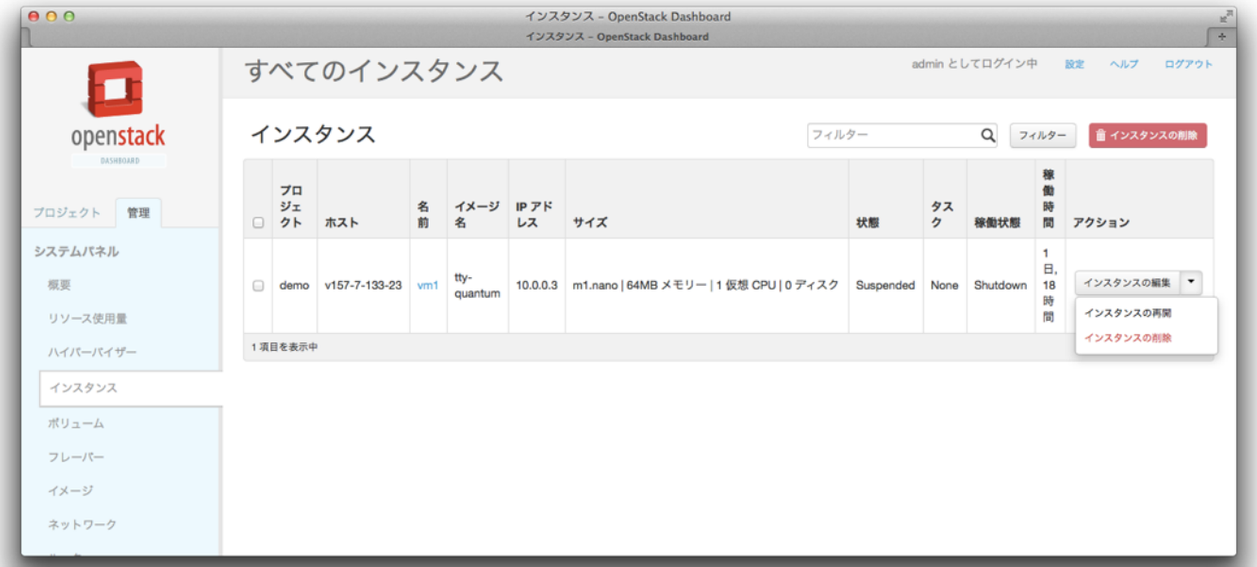
  https://github.com/yosshy/openstack-ansible



OpenStackの操作イメージ

- コマンドラインインターフェイスからの操作
  コマンドラインが提供されているため作業のスクリプト化も可能
- WebUIからの操作
  バージョンアップを重ねる毎に安定度と機能を増している
- 外部APIを利用したソフトウェア制御
  REST-APIが提供されており、各コンポーネントの機能をソフトウェア制御可能

クラウド全体のリソース使用状況も一目瞭然



仮想マシンに対するさまざまな操作が可能

日本OpenStackユーザ会

- OpenStackに関して日本語で質問したい!
- 日本語のドキュメントが欲しい!
- 独りで挫けそう…そんなアナタはコチラまで。
  - GoogleGroup

    http://groups.google.com/group/openstack-ja
  - ユーザ会Webサイト

    http://openstack.jp/

Step-2 まとめ

- インフラ構築の自動化はStep-1で行ったものとは規模が違う。導入には相当の覚悟と時間が必要
- 既存のインフラ構築手法を自動化する場合は、元々のシステムの素性が大きく影響する
  - 様々な理由で「人」が作業することを前提としたメンテナンスラインしか存在していない場合もある
  - IPMI・NETCONFなどHW機器のソフトウェア制御への対応状況もまちまち。コスト見合いで効率的な手法が採用できない場合もある
- 自動化に対する労力が大きい分、効果も絶大です。がんばる価値あり

Infrastructure as Code

Step 3
～協調動作～

Step-3: さまざまな仕組みの協調動作[1]

Step-2で採用した大きな仕組みには1つの共通点がある

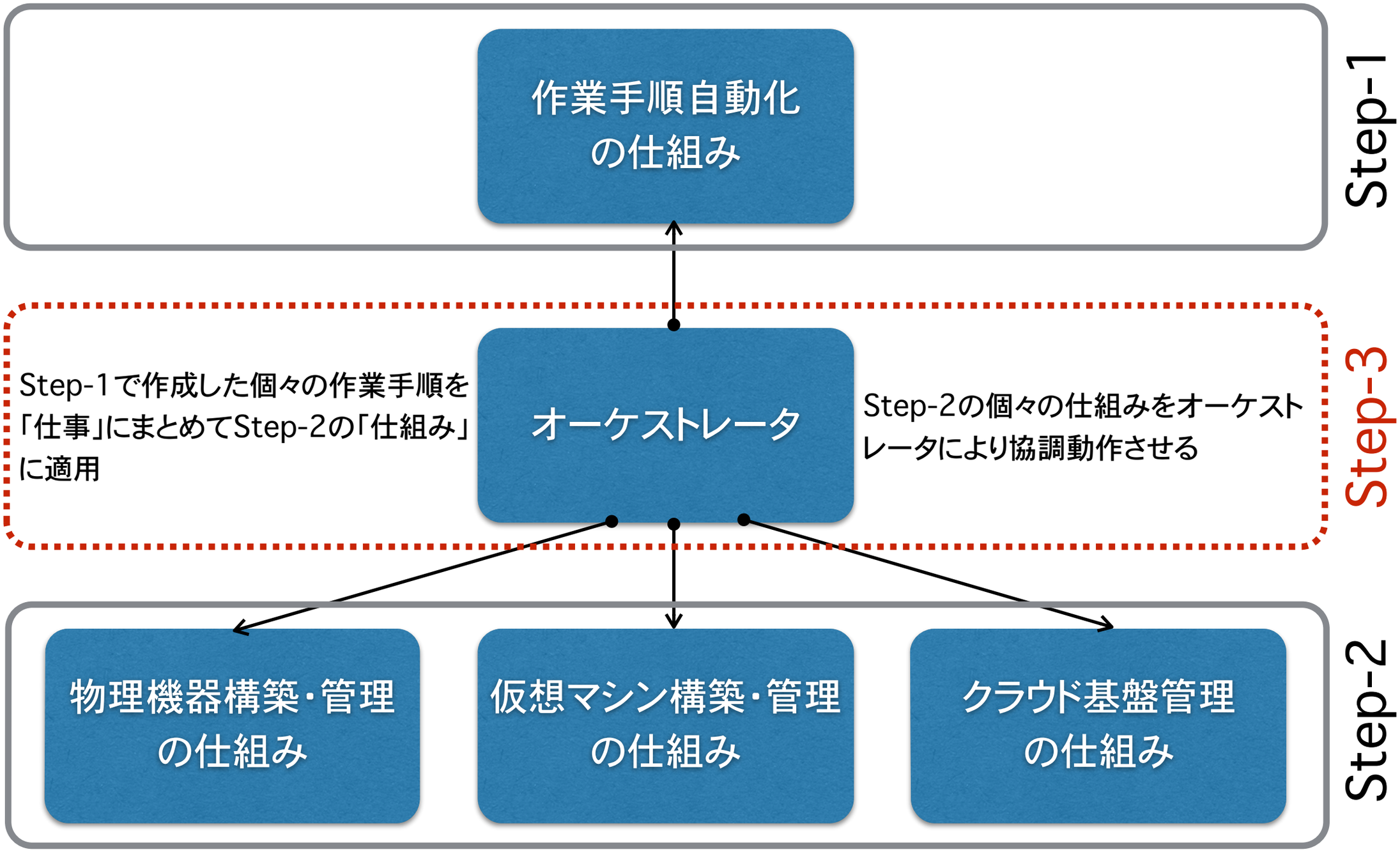
外部からソフトウェアによる
制御可能である

外部APIを利用して、ソフトウェア(オーケストレータ)が個々の仕組みを協調動作させるのがStep-3のゴール

Step-3 個々の仕組みの協調動作

Step-3: さまざまな仕組みの協調動作[2]

オーケストレータによる協調動作



Step-3: さまざまな仕組みの協調動作[3]

世の中には、さまざまなタイプのオーケストレータがありますが、ここでは大きく2つに分けてみます。

仮想化基盤だけでなく物理基盤も制御する必要があるのであれば、選択肢は従来型のものということになります（現時点では）。

- スクリプトなど従来のツールの延長線上にあるもの
  - Puppet
  - Chef
  - Capistrano
  - Ansible（AWS） ※今回ご紹介します
- さまざまな出自の異なるクラウド基盤の制御とAPI共通化に注力したもの
  - Apache libcloud
  - Deltacloud
  - Rightscale
  - SCALR

Ansibleによる構成管理

- What?/Why?
  - Pythonで書かれた*作業*だけなく*仕事*を自動化するためのツール
  - OS・ミドルウェア・アプリケーションのインストール・設定を自動化可能
  - 作業の一連の流れをPlaybookとして1つにまとめることができる
  - 多数の外部モジュールが標準・オプションで提供されている
  - 自動化した作業の冪等性が担保されている(担保されていない外部モジュールもある)
  - push型でエージェントレス。操作対象ノードにSSHでログインできれば最低限ど動作するため導入の敷居が低い
  - 外部から制御可能なAPIを持っている(現状はAWXを利用したREST API)
- Who?/When?/Where?
  - michael.dehaanさん(Ansible Works)
  - 公式サイト: http://www.ansibleworks.com/
  - ソースコード: https://github.com/ansible/ansible
  - githubのtagによるとv0.01のリリースは2009年
  - 現時点(2013-11)で最新の安定版はv1.4

Ansibleの構成要素

- module

  サービスの起動停止などansibleに行わせる作業を外部モジュールとして管理する

- playbook

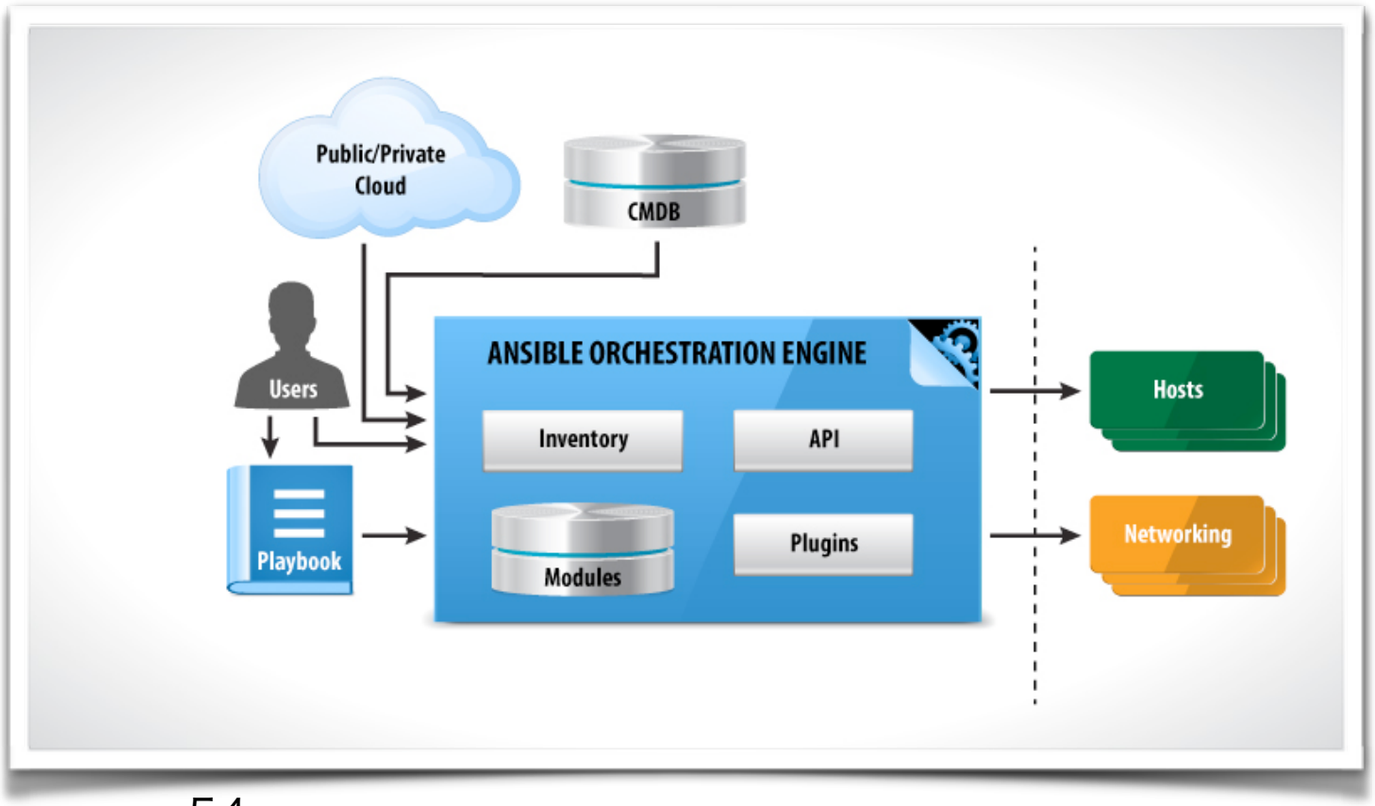
  作業(module)の一連の流れをまとめたもの

- plugin

  moduleの実行失敗時のcallbackなどansibleそのものの機能を提供する

- inventory

  操作対象ノードに関する情報が記録されている



Ansibleの導入から稼働まで(ubuntu12.04lts編 [1])

1. 必要となるパッケージ群をダウンロード

12.04のパッケージ版はv1.1と古く、OpenStack用のモジュールが提供されていないためgithubからソースコード(v1.3.4)を取得してインストールする

```
$ sudo apt-get install -y git python-pip python-virtualenv
```

2. virtualenv環境構築してAnsibleをインストール

1. virtualenv環境構築

ansibleをOS標準とは別のモジュールパスにインストールする

```
$ virtualenv app
New python executable in ansible/bin/python
Installing distribute....done.
Installing pip...done.
$ source app/bin/activate
(app) $
```

2. ansibleが依存するモジュール群をインストール

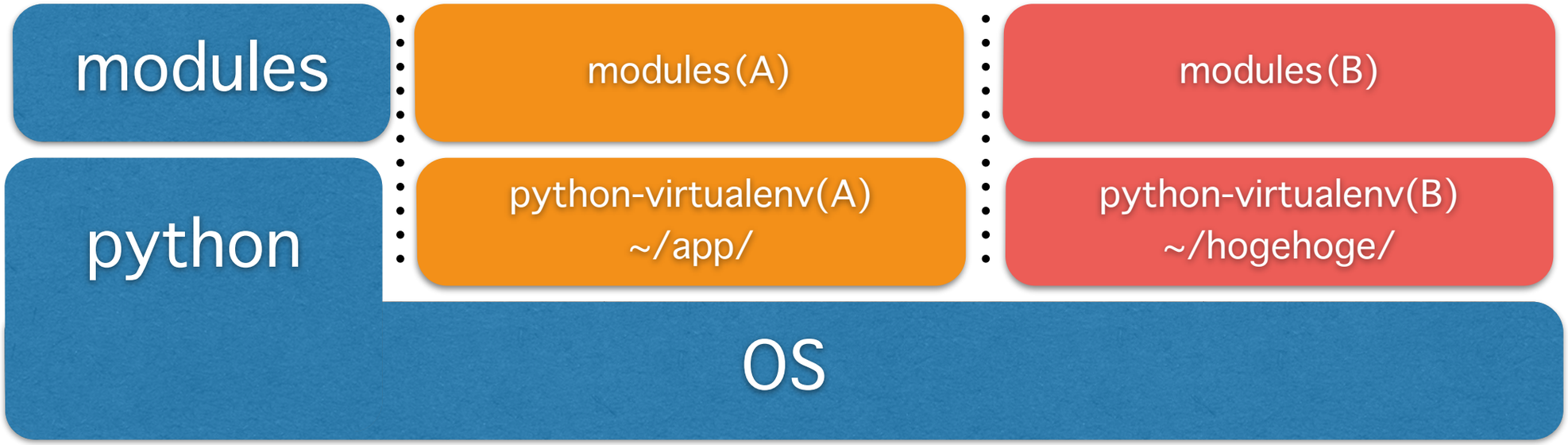
```
(app) $ pip install PyYAML jinja2
```

Ansibleの導入から稼働まで(ubuntu12.04lts編 [2])

3. githubからansibleを取得してインストール
   - v1.3.4をcheckoutする
   - この例ではモジュールはapp/share/ansible/にインストールされる

```
(app) $ mkdir src && cd src
(app) $ git clone https://github.com/ansible/ansible.git
(app) $ cd ansible
(app) $ git checkout -b v1.3.4 v1.3.4
(app) Switched to a new branch 'v1.3.4'
(app) $ python setup.py build
(app) $ python setup.py install
```

<pythonのvirtualenv環境>



virtualenvの外側に影響を与えない

Ansibleの導入から稼働まで(ubuntu12.04lts編 [3])

3. 動作確認

1. インベントリファイルを作成

   インベントリには操作対象となるホストのIPアドレスを記載する

   ※他にも様々な記載方法が可能

```
(app) $ echo “192.168.100.110” > hosts
```

2. pingモジュールで動作確認(successとなることを確認する)

```
(app) $ ansible -i hosts all -u foo -k -m ping
SSH password: ********
192.168.100.110 | success >> {
    "changed": false,
    "ping": "pong"
}
```

-i: インベントリファイル名

-u: 対象ノードにsshログインするユーザ名

-k: 実行時にsshログインパスワードを入力するインタラクションを発生させる

-m <モジュール名>

Ansibleの導入から稼働まで(ubuntu12.04lts編 [4])

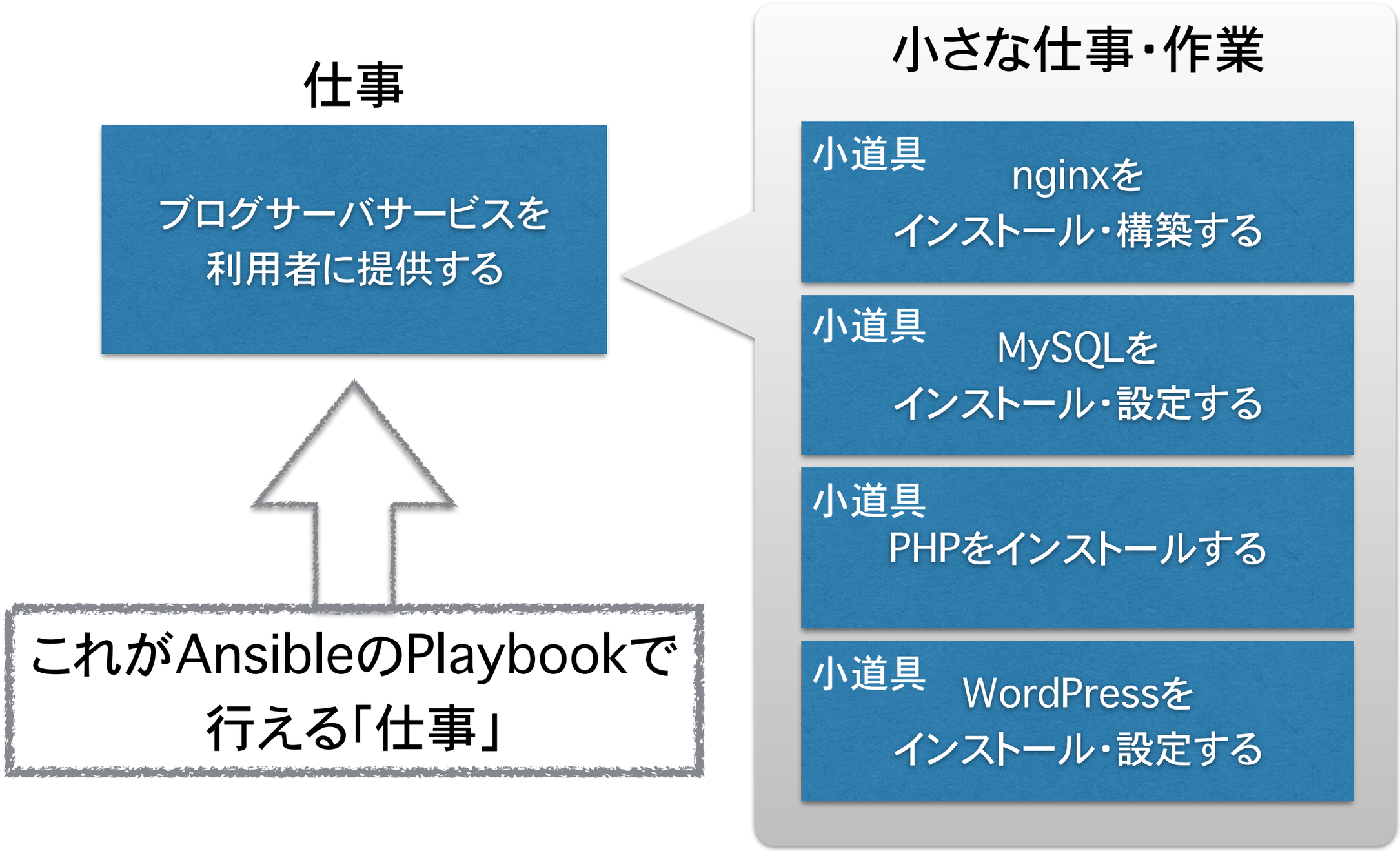
3. setupモジュールで操作対象サーバの情報を取得してみる

   取得した情報は環境変数としてPlaybook内で利用可能

```
(app) $ ansible -i hosts all -u foo -k -m setup
SSH password: ********
172.16.8.130 | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "192.168.122.1",
            "172.16.8.130",
            "169.254.169.254"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::20c:29ff:fe8e:2665",
            "fe80::883d:a8ff:fead:cfaf"
        ],
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "07/31/2013",
        "ansible_bios_version": "6.00",
        "ansible_br_ex": {
            "active": true,
[省略]
```

AnsibleのPlaybookで仕事をさせてみる(1)

apacheやmysqlのインストールを行い、適切に設定してサービスを起動する。など1つ1つの「作業」をまとめて、「ブログサーバを構築して利用者に提供する」といったように抽象化したものを、ここでは「仕事」と呼ぶことにすると…



AnsibleのPlaybookで仕事をさせてみる(2)

Playbookのディレクトリ構造

```
playbook名/
    group_vars/
        all
    roles/
        小さな仕事-A/
            tasks/
                main.yml
            handlers/
                main.yml
            templates/
                foo.conf.j2
                bar.conf.j2
                baz.conf.j2
            file/
                hogehoge.conf
        小さな仕事-B/
        小さな仕事-C/
    site.yml
```

たとえば”wordpress-release”など「仕事」の名前

Playbook内でで利用する環境変数を定義するファイル

mysqlの設定・起動などの「小さな仕事」の単位をroles内で定義

実際に行う作業を定義

tasks内の作業結果をトリガーとして実施する必要のある動作を定義。例えば、tasksでmy.cnfの修正作業を実施場合に必要となる「mysqldの再起動」などを定義する

my.cnfなどの設定ファイルのテンプレートを置く。テンプレート中に{{ 環境変数 }} のように書くことでjinja2でパラメータの置き換えが可能

対象サーバにそのまま置けば良いファイルを定義

Playbookが含む小さな仕事(roles内の仕事)のリストを定義

AnsibleのPlaybookで仕事をさせてみる(3)

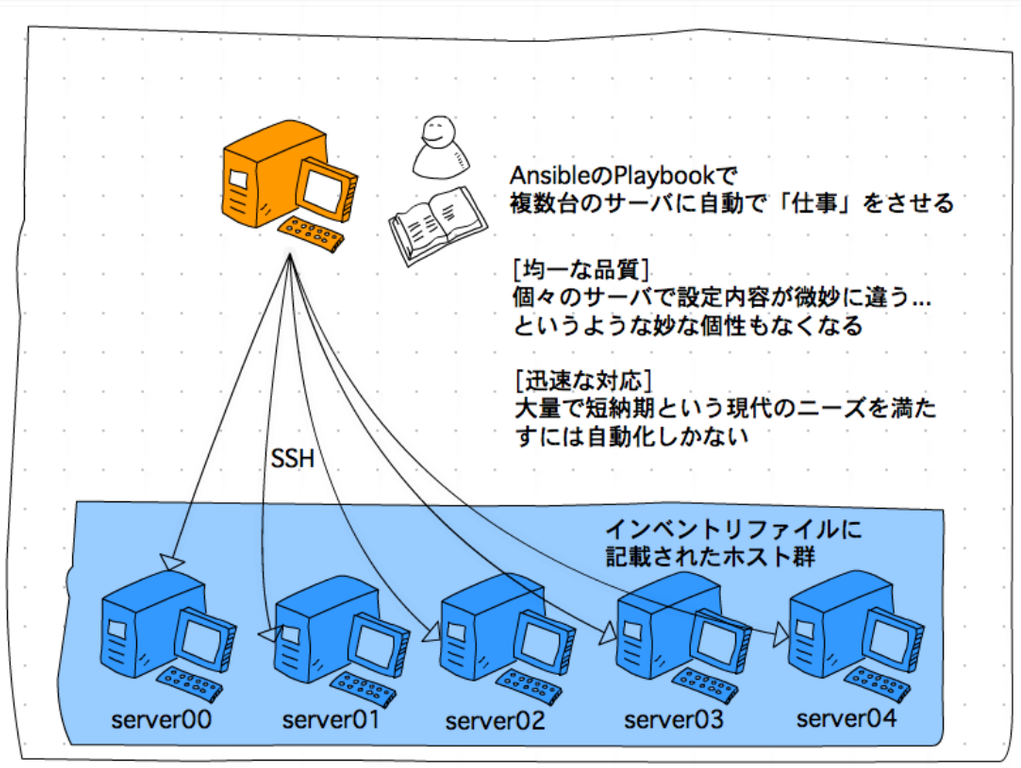
ブログサーバを構築して提供するPlaybookを実行してみる

```
$ ansible-playbook -u root -k -i hosts site.yml
SSH password: ********

PLAY [Install Wordpress, MySQL, Nginx, and PHP-FPM] ***************************

GATHERING FACTS ***************************************************************
ok: [192.168.100.110]
[省略]
TASK: [Install Mysql package] *************************************************
changed: [192.168.100.110] => (item=mysql-server,MySQL-python,libselinux-python,libsemanage-python)

TASK: [Create Mysql configuration file] ***************************************
changed: [192.168.100.110]
[省略]
PLAY RECAP ********************************************************************
192.168.100.110            : ok=25   changed=24   unreachable=0    failed=0
```



Ansibleで「すぐにできること」・「工夫しないとできない」こと

Step-2の構築・管理の自動化を行う「仕組み」に対してAnsibleができること、工夫しないとできないことは以下の通り

- すぐにできること(OpenStackを操作する)
  - Nova (仮想マシンの作成・削除)
  - Glance (OSイメージ登録・削除)
  - Neutron (仮想ネットワーク/ルータの作成・削除・IPアドレス管理)
  - Keystone (ユーザ・テナント・ロールの作成・削除)
  - Ansible標準の外部モジュール群:

    http://www.ansibleworks.com/docs/modules.html

- 工夫しないとできないこと(外部モジュールを開発する)

  Cobbler/JUNOSの設定など標準で提供されない仕組みは外部モジュールを自前で用意することになる

  1)モジュール呼び出しの引数はテキストファイルで渡されるため、これを読み込む手段があること

  2)実行結果はJSON形式で標準出力する必要があるためJSONを扱えること

  開発言語は以上の2つの条件を満たしていればOK

Step-3 まとめ(協調動作)

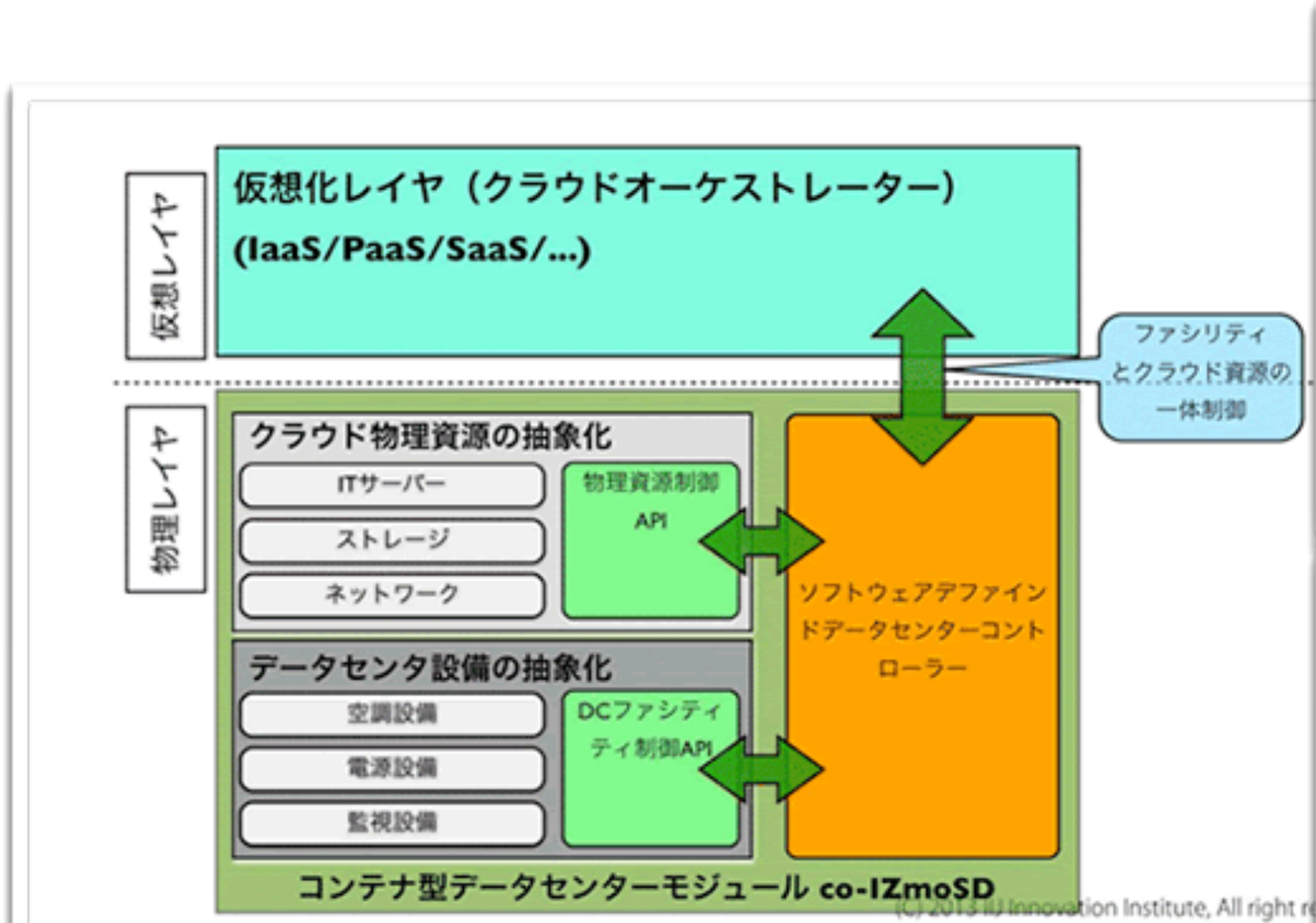
- Step-1で行った個々の作業の自動化、Step-2で行ったインフラ構築とクラウド基盤管理の自動化をベースに「仕事」をコード化する
- 個々の仕組みを協調して動作させることで、「人」が行う単純作業を劇的に減らせる
- オートスケールのようなモダンな機能を実現するための土台ができる
- Step-3がある程度確立できれば、システムとしての素性も良くなり、この先に待ち受けている継続的デリバリや自律化のような、システムの高度な利用方法に対応する準備ができる

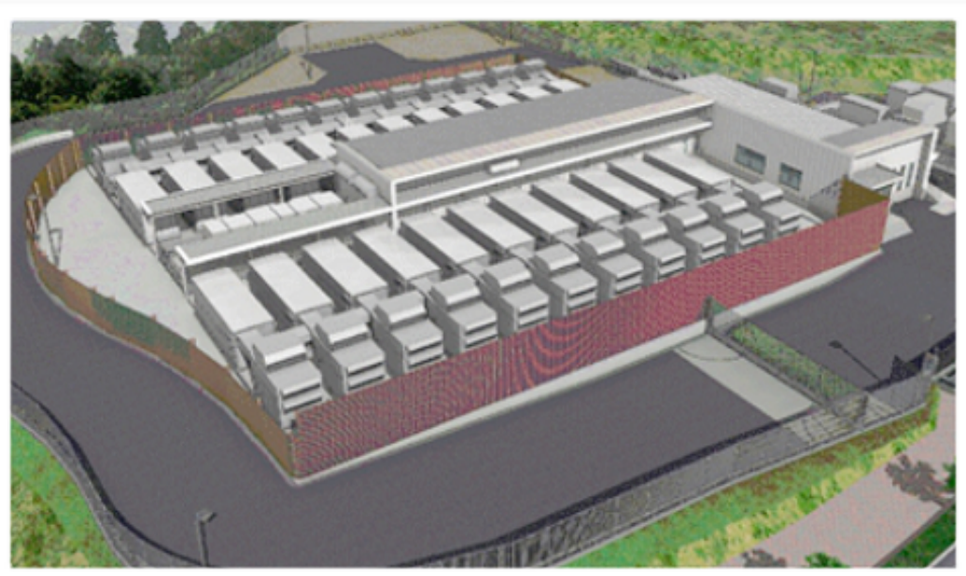
Infrastructure as Code
Step 4
～未来に向けて～

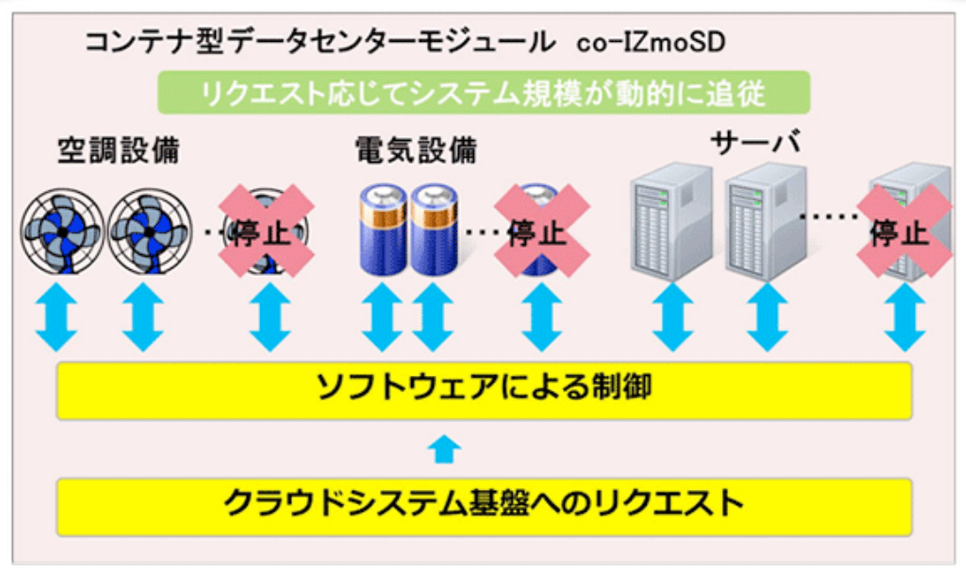
最近はこんなことやってます

- ソフトウェア制御可能なデータセンターの研究開発

URL: http://www.iij.ad.jp/news/pressrelease/2013/1010.html







今回ご紹介したStep1-2-3で利用したツール類

- Step-1-2-3で利用したツール類
  - cobbler

    http://www.cobblerd.org/
  - NETCONF and junos

    http://goo.gl/qCTVgw
    https://github.com/Juniper/net-netconf
    http://ncclient.grnet.gr/
  - OpenStack

    http://openstack.org/
    - devstack
      http://devstack.org/
    - RDO
      http://jp-redhat.com/openstack/rdo/
  - Ansible

    http://www.ansibleworks.com/

今回ご紹介したStep1-2-3で利用したツール類

- Step-1 - 2 -3 で紹介したツール類
  - cobbler

    http://www.cobblerd.org/
  - NETCONF & junos

    https://github.com/Juniper/net-netconf
    http://ncclient.grnet.gr/
  - OpenStack

    http://openstack.org/
    - devstack

      http://devstack.org/
    - RDO

      http://jp-redhat.com/openstack/rdo/
  - Ansible

    http://www.ansibleworks.com

本セッション全体のまとめ

- Infrastructure as Codeの一例を世代を追って紹介しました
- まずは小さなことから始めて徐々に自動化をしていく
- 小さな作業をまとめて仕事として抽象化する
- これまでバラバラに作ってきた仕組みをソフトウェア(人ではない)で協調動作させる
- 仕事の抽象化とシステムを構成管理するための道具は近年急速に充実してきている。自分たちの仕事にあった道具を選ぼう(Puppet/Chef/Ansible…)
- 常に「もうひと工夫」を意識する
- くどいようだが「人」に頼らない仕組みを作る
  - システムの素性も大切。最初から人間がオペレーションすることを前提としない設計
  - パーツとして使うツール類は外部プログラム(構成管理システムなど)からコントロール可能なAPIを持っていることが重要

ご清聴ありがとうございました